Week 1 - Report

Group 3

File I/O wrappers

Contributors - Yashvardhan Toshniwal, Kritang Kothari, Daksh Sehra

File I/O wrappers are custom implementations designed to enhance the functionality, security, and performance of standard file input/output operations in the Linux kernel. By wrapping around system calls such as `open`, `read`, `write`, and `close`, these wrappers provide additional features like logging, access control, error handling, and performance optimization.

Here are the File I/O wrappers we plan to implement in this project** :

Logger

- **Basic Idea**:
  - A logging mechanism to track file access patterns, including operations like `open`, `read`, `write`, and `close`.
- **Advantages**:
  - Helps in debugging.
  - Monitors application behavior.

Safe Read

- **Basic Idea**:
  - A secure way to read files that checks if the file is empty before reading.
  - Ensures that read operations do not exceed the file size.
- **Advantages**:
  - Gives warning reading from empty files.
  - Avoids reading beyond the end of the file.
  - Ensures robust error handling for read operations.

Safe Open

- **Basic Idea**:
  - A secure way to open files with error handling to ensure they are opened successfully.
  - Automatically handles read/write mode specifications.
  - Give some information about what went wrong eg. Incorrect file names for `open`.
- **Advantages**:
  - Prevents common file handling errors.
  - Ensures robust error handling.

Access Control and Permissions Enforcement Wrapper

- **Basic Idea**:
  - Ensures that only authorized users can access certain files or functionalities by wrapping system calls like `open`.
- **Advantages**:
  - Enhances security by preventing unauthorized access.

Rate-Limiting

- **Basic Idea**:
  - Controls the rate at which file operations can be performed by wrapping system calls like `open` and `read` to track and limit access rates.
- **Advantages**:
  - Protects against abuse.
  - Ensures fair resource usage.
  - Prevents denial-of-service attacks by throttling excessive access.

Buffer

- **Basic Idea**:
  - A temporary storage area for data that can be used for buffering write operations.
  - Auto-flushes after a set size limit.
- **Advantages**:
  - Improves performance by reducing the number of write operations.
  - Prevents data loss.
  - Provides a mechanism for undo functionality.

Custom Caching Mechanism

- **Basic Idea**:
  - Stores frequently accessed data in memory to reduce disk I/O for repeated reads.
  - Uses an LRU (Least Recently Used) cache strategy. (Up for discussion with TA)
- **Advantages**:
  - Reduces latency.
  - Improves application performance.
  - Minimizes disk access for frequently read files.

Master Wrapper

- **Basic Idea**:
  - A comprehensive wrapper that integrates all other wrappers.
  - Gives a front-end interface to set flags to select which wrappers to be used.
- **Advantages**:
  - Simplifies the management of individual wrappers.
  - Ensures cohesive functionality.
  - Provides a single point of control for various file I/O enhancements.

System Call Wrappers for Memory Management

Contributors - Raadhes Chandaluru, Dev Mehta, Mith Jain, Rohan Bagatti

`safe_mmap`: Memory Mapping with Allocation Tracking

- **Basic Idea**:
  - Wraps `mmap` to handle errors, track allocations, and enable allocation reuse.
  - Logs usage statistics to aid in performance monitoring and debugging.
- **Advantages**:
  - Enhanced error handling: If `mmap` fails, it reports errors and terminates safely.
  - Improves debugging and performance analysis through detailed logging of memory usage.

`safe_munmap`: Safe Memory Unmapping

- **Basic Idea**:
  - Provides a safer way to unmap memory while maintaining allocation tracking.
  - Checks for errors from `munmap` and prints descriptive messages if it fails.
- **Advantages**:
  - Prevents undefined behavior by ensuring `munmap` is only called on valid memory regions.
  - Makes memory management more robust by integrating error handling and tracking updates.

`debug_malloc` and `debug_free`: Debugging Dynamic Memory

- **Basic Idea**:
  - Tracks `malloc` and `free` calls to detect memory leaks and misuse.
  - Provides detailed output for every allocation and deallocation, helping developers trace memory usage.
- **Advantages**:
  - Simplifies memory debugging by automatically logging every memory operation.
  - Detects and pinpoints memory leaks, reducing the risk of out-of-memory errors in long-running programs.
  - Provides clear visibility into memory usage patterns, aiding performance tuning.

`robust_read`: Resilient Reading from Files

- **Basic Idea**:
  - Wraps `read` to handle interruptions (`EINTR`) automatically.
  - Implements a loop that retries `read` calls until successful or until an error other than `EINTR` occurs.
- **Advantages**:
  - Improves reliability of file I/O operations by ensuring data is read completely.
  - Reduces error-handling complexity for the user, leading to cleaner and more robust code.

`safe_fork`: Fork with Resource Cleanup

- **Basic Idea**:
  - Ensures resources like file descriptors and logs are properly handled during `fork`.
  - Flushes standard I/O buffers (`stdout`, etc.) to avoid duplication in both parent and child processes.
  - Provides a cleanup hook (`on_fork_cleanup`) to allow custom resource cleanup actions.
- **Advantages**:
  - Prevents inconsistent logging or file writes caused by unflushed buffers.
  - Reduces risk of resource contention or duplication between parent and child processes.
  - Provides flexibility to extend cleanup for user-defined resources.

`tracked_open` **and** `close_all_files`**: File Descriptor Management**

- **Basic Idea**:
  - Manages open file descriptors to prevent leaks and facilitate bulk cleanup.
  - Tracks file descriptors opened via `open` in a static array (`open_files`).
  - Provides `close_all_files` to iterate over and close all tracked file descriptors, ensuring no leaks.
- **Advantages**:
  - Simplifies resource management, particularly in programs with many open files.
  - Prevents resource leaks by enforcing proper closure of all file descriptors at program exit or error recovery.
  - Improves program stability by preventing exhaustion of file descriptor limits.

`auto_resize_buffer`**: Dynamic Buffer Resizing**

- **Basic Idea**:
  - Automatically resizes a buffer when more space is needed.
  - Implements an exponential growth strategy (`new_size * 2`) to minimize the number of reallocations.
  - Ensures that reallocation failures are properly handled with error reporting and program termination.
- **Advantages**:
  - Reduces overhead associated with frequent resizing, leading to better amortized performance.
  - Provides a robust mechanism for handling dynamic data structures like strings or arrays without manual size management.
  - Makes user code simpler and more maintainable by abstracting away the resizing logic.

`aligned_mmap` **and** `aligned_munmap`**: Aligned Memory Allocation and Deallocation**

- **Basic Idea**:
  - These wrappers ensure that memory is allocated and freed with specific alignment requirements.
  - `aligned_mmap`: Allocates memory with the specified alignment by reserving extra space and adjusting the returned pointer to be aligned.
  - `aligned_munmap`: Deallocates the aligned memory while ensuring that the correct base address is passed to `munmap`.
- **Advantages**:
  - Ensures compatibility with systems or libraries requiring aligned memory (e.g., for SIMD instructions or hardware page alignment).
  - Improves performance by reducing the overhead of misaligned memory accesses.
  - Simplifies deallocation by correctly identifying the base address of the allocated region, preventing potential errors.

`safe_execvp`: Secure Execution of External Programs

- **Basic Idea**:
  - A safer wrapper around `execvp` that validates input arguments and handles common errors gracefully.
  - Validates the program path and argument list before invoking `execvp`.
  - If `execvp` fails (e.g., due to missing executables or permission errors), it logs detailed error messages without terminating the calling process.
- **Advantages**:
  - Improves reliability by catching and reporting errors in program execution.
  - Enhances security by ensuring inputs are validated, reducing the risk of unexpected behavior or crashes.
  - Provides better user experience through informative error messages, aiding debugging of execution failures.

Process Wrappers

Contributors Contributors - Aditya Srivastava, Aditya Jain, Harsh Vardhan Daga, Shreyanshu Gurjar

Process Pool Manager

- **Wrapper description**

  This Process Pool Manager is a wrapper that manages a pool of worker processes. It allows tasks to be distributed efficiently across multiple processes, much like a thread pool but using processes for better isolation and utilization of multi-core systems.
- **Basic Idea**
  - **Dynamic Task Queue:** Accepts tasks dynamically and distributes them to workers (different child processes).
  - **Inter-Process Communication (IPC):** Uses pipes for communication between the parent and worker processes.
  - **Task Scheduling:** Distributes tasks to available workers according to the priority of the tasks.
  - **Graceful Shutdown:** Ensures all tasks are completed before shutting down.
- **Advantages**
  - Reduces overhead by reusing existing worker processes, avoiding frequent process creation and destruction.
  - Maximizes CPU core usage by running processes in parallel.
  - Provides a simple interface for passing tasks and results between the parent process and workers and ensures efficient communication, minimizing bottlenecks.

Process Graph Deadlock Detection and Mitigation through System Call Wrappers

- **Wrapper description**

  To handle potential deadlocks, we create wrappers for system calls related to process creation (`fork()`) and resource allocation. These wrappers build a **process-resource dependency graph** dynamically, detecting circular waits and thus potential deadlocks.

- **Basic Idea**

  - **Track Dependencies**: Whenever a process requests or releases a resource, we record the interaction in a graph.
  - **Detect Cycles**: A **cycle detection** algorithm (like DFS) is run periodically to check if the process graph contains circular dependencies.
  - **Handle Deadlocks**: On detecting a deadlock, one of the processes is killed or rolled back to break the cycle.

- **Advantages**

  - **Real-Time Detection**: Enables continuous monitoring of process-resource dependencies, allowing immediate detection of potential deadlocks.
  - **Minimal Overhead**: Wrappers efficiently integrate with existing system calls, introducing minimal performance impact during process execution.
  - **Enhanced Debugging**: Provides clear insights into process behavior and dependencies, aiding in debugging and improving system reliability.
  - **Scalability**: Can be extended to large-scale systems with complex resource dependencies due to its modular and systematic approach.

Process Cloaking Wrapper

- **Wrapper description**

  To create child processes that are hidden from common process monitoring tools like `ps` and `top` by isolating them in separate namespaces, detaching from terminals, and modifying process credentials.

- **Potential Uses**

  - Running background services or maintenance tasks that do not need to be visible to users.
  - Security applications, where certain processes need to run without being easily detected.

- **Advantages**

  - **Enhanced Process Privacy**: Prevents specific processes from being monitored through common tools like `ps` or `top`, improving confidentiality.
  - **Increased Security**: Reduces the risk of detection by attackers or unauthorized users, useful for sensitive tasks or security-focused applications.

- **Isolation for Background Tasks**: Ideal for running maintenance or background services without interference or visibility to regular users.
- **Session and Namespace Detachment**: Effectively isolates the cloaked process from its parent session and namespace, making it more difficult to trace or interfere with.

Wait_for_all_children wrapper

- **Wrapper description**

  The wrapper `Wait_for_all_children` when called with a parent `pid`, it blocks the parent process until all the child processes exit.

- **Basic Idea**

  - The file `/proc/<PID>/task/<TID>/children` contains the child PIDs of a given thread or process.
  - Iterate over child processes and call `waitpid()`.

- **Advantages**

  - **Prevent Zombie Processes:** Ensures no child process is left in a zombie state. A process doesn't need to call `waitpid()` for every child process it creates as this wrapper automates calling `waitpid()` on all child processes.

Comprehensive IPC Logging Wrappers

- **Wrapper description**

  A collection of logging wrappers for various IPC mechanisms such as pipes, message queues, shared memory, semaphores, and signals. These wrappers log key information about the system calls without altering the underlying application logic. The logs provide useful insights into the operations, parameters, and results of the IPC calls, improving debugging and monitoring.

- **Advantages**

  - **Comprehensive Auditing**: Provides a complete log of all Inter-Process Communication (IPC) events, enhancing traceability and accountability in the system.
  - **Security Monitoring**: Helps in identifying unauthorized or suspicious IPC activity, improving overall system security.
  - **Debugging Aid**: Facilitates debugging by offering detailed insights into process interactions and message exchanges, making error tracking easier.
  - **Post-Incident Analysis**: Enables forensic analysis of system behavior after failures or security breaches by maintaining a history of IPC exchanges.

Priority Enforcement wrapper

- **Wrapper description**

  The Priority Enforcement Wrapper extends the `fork()` system call to set CPU and I/O priorities for child processes. This helps ensure that high-priority tasks receive sufficient resources while low-priority tasks don't hinder overall system performance (Up for discussion with TA).

- **Basic Idea**

  - After calling fork(), the wrapper sets the CPU and I/O priorities for the child process.
  - CPU priority is set using `setpriority` and `nice`.
  - I/O priority is set using a system command with `ionice`.

- **Advantages**

  - **Controlled Resource Usage:** Lower-priority tasks run without impacting high-priority applications.
  - **Dynamic Resource Allocation:** Easily adjust priorities at runtime to adapt to system load.

Process Monitoring and Logging Wrapper

- **Wrapper description**

  This wrapper monitors key system calls such as fork(), execvp(), and waitpid(), logging the status and outcome of each operation to a log file.

- **System Calls Used**

  - `fork()`: Creates a new process.
  - `waitpid()`: Waits for a specific child process to finish.
  - `execvp()`: Runs a new program, replacing the current process with the new program.

- **Advantages** :

  - Helps in debugging.

Custom Waitpid Wrapper

- **Wrapper description**

  The **Custom Waitpid Wrapper** extends the `waitpid()` system call to enhance the monitoring and handling of child processes. This wrapper allows the parent process to track the status of long-running child processes, implement timeouts to prevent indefinite blocking, and provides detailed information on the exit status of the child process.

- **Basic Idea**

  - After invoking `waitpid()`, the wrapper tracks the child process's status, handling cases of long execution times or abnormal termination.
  - It implements timeouts for processes that run longer than expected, avoiding indefinite blocking.
  - Provides detailed exit statuses for child processes, indicating whether they terminated normally, were signaled, or stopped.

- **Advantages**

  - **Improved Monitoring:** Tracks and provides detailed insights into child process statuses.
  - **Timeout Handling:** Implements timeouts to avoid indefinite blocking of long-running processes.
  - **Enhanced Debugging:** Tracks process status and provides useful information to detect abnormal terminations or timeouts.

Mutex Synchronization with Ownership Tracking

- **Wrapper description**

  The **Mutex Synchronization with Ownership Tracking Wrapper** is designed to ensure thread synchronization in a multi-threaded environment. This wrapper allows threads to safely access critical sections by using mutexes while also tracking the ownership of the mutex. It prevents misuse and helps avoid deadlocks by keeping track of which thread currently holds the mutex.

- **Basic Idea**

  - Mutexes are used to synchronize threads, ensuring that only one thread can access a critical section at a time.
  - The wrapper adds an ownership tracking feature to keep track of which thread currently owns the mutex, providing better control and avoiding potential misuse.

- **Advantages**

  - **Ownership Tracking:** Tracks the thread that holds the mutex, preventing misuse and deadlocks.
  - **Debugging Support:** Provides insights into which thread currently holds the mutex, making it easier to diagnose synchronization issues.

Zombie Process Logger

- **Wrapper description**

  This wrapper extends the `kill()` and `waitpid()` system calls to provide enhanced process management and logging. This wrapper is designed to log any zombie processes when these syscalls are invoked. It tracks the lifecycle of child processes, ensuring that no zombies remain after termination, and logs relevant information for debugging and process management.

- **Basic Idea**

- The wrapper intercepts calls to `kill()` and `waitpid()` to manage the termination of processes.
- When a process is terminated using `kill()`, it ensures that the process is properly cleaned up and prevents it from becoming a zombie.
- After the `waitpid()` syscall is invoked, the wrapper checks for any remaining zombie processes and logs their details.
- This wrapper ensures that no zombie processes are left behind by handling both the termination and the cleanup process in a single sequence.

- **Advantages**
  - **Zombie Process Logging:** Logs details about any zombie processes, making it easier to track and debug process termination issues.
  - **Process Cleanup:** Ensures that processes are properly terminated, avoiding the creation of zombie processes.
  - **Debugging Support:** Provides useful information in the logs to help identify issues related to process termination and zombie processes.

The current list is tentative, and the final set of wrappers we create may differ slightly from this version. Some implementation details are also provided but it is only tentative. Real implementation can vary upon final requirements.